

I web services

Introduzione ai web services

Un **web service** (detto anche API = Application Programming Interface) è una applicazione in esecuzione su un **web server** che, a fronte di una richiesta http, restituisce un insieme di dati (in formato XML o JSON).

Il concetto di **web service** nasce negli anni 2004-2005 quando ci si rese conto che ricaricare ogni volta una intera pagina HTML risultava molto oneroso, mentre sarebbe stato molto più conveniente (e anche molto più semplice) ricaricare di volta in volta soltanto i dati necessari a seconda dei contesti.

La tecnologia utilizzata dei browser per accedere ai dati esposti da un **Web Service** si chiama **AJAX (Asynchronous JavaScript And XML)**

Introduzione ai web services

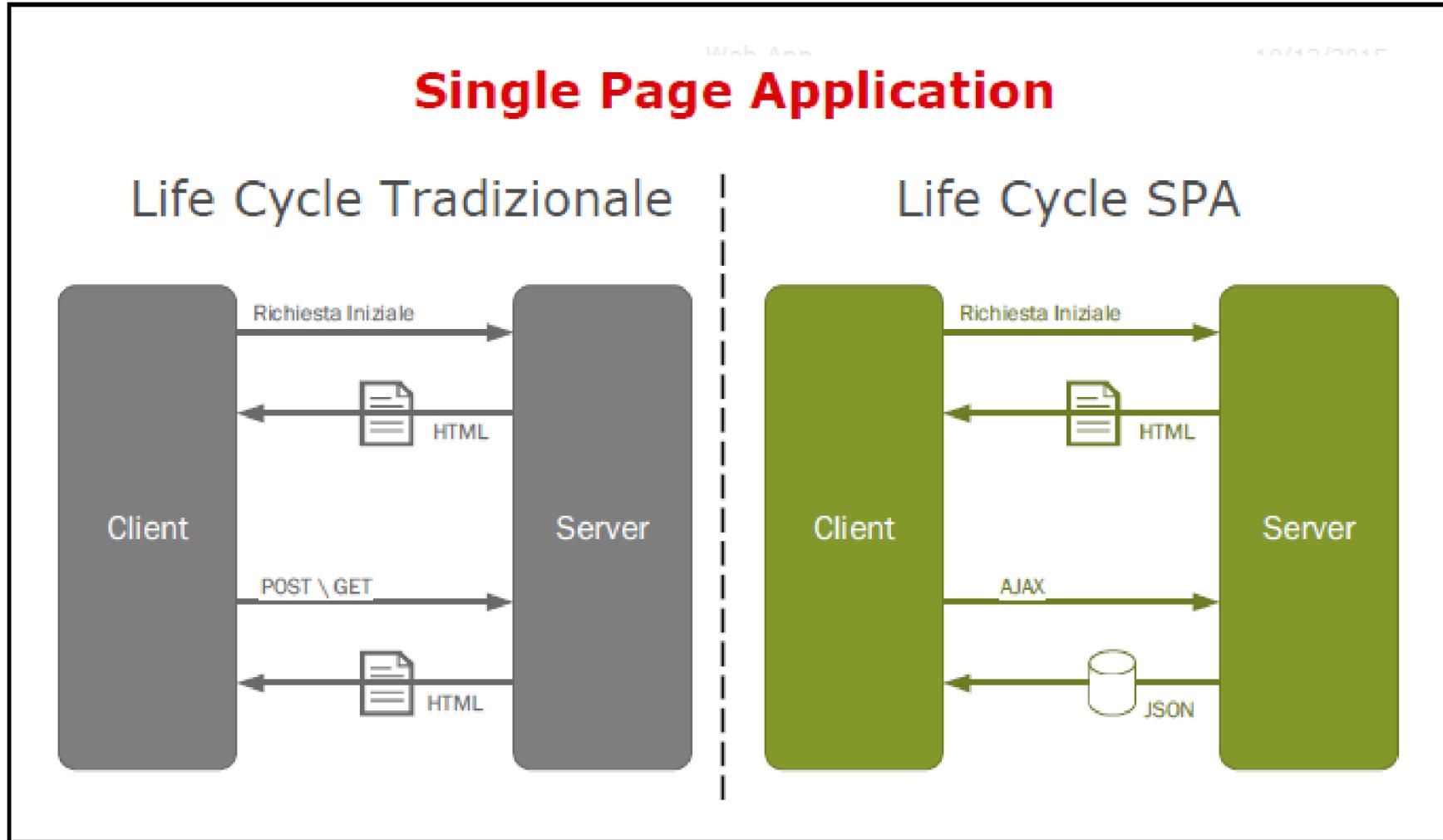
Un **web service** (detto anche API = Application Programming Interface) è una applicazione in esecuzione su un **web server** che, a fronte di una richiesta http, restituisce un insieme di dati (in formato XML o JSON).

Il concetto di **web service** nasce negli anni 2004-2005 quando ci si rese conto che ricaricare ogni volta una intera pagina HTML risultava molto oneroso, mentre sarebbe stato molto più conveniente (e anche molto più semplice) ricaricare di volta in volta soltanto i dati necessari a seconda dei contesti.

La tecnologia utilizzata dei browser per accedere ai dati esposti da un **Web Service** si chiama **AJAX (Asynchronous JavaScript And XML)**

Insieme ad Ajax nasce anche il concetto di **SPA Single Page Application**. Il client effettua una unica richiesta iniziale di pagina e poi provvede ad aggiornarla tramite richieste dati successive.

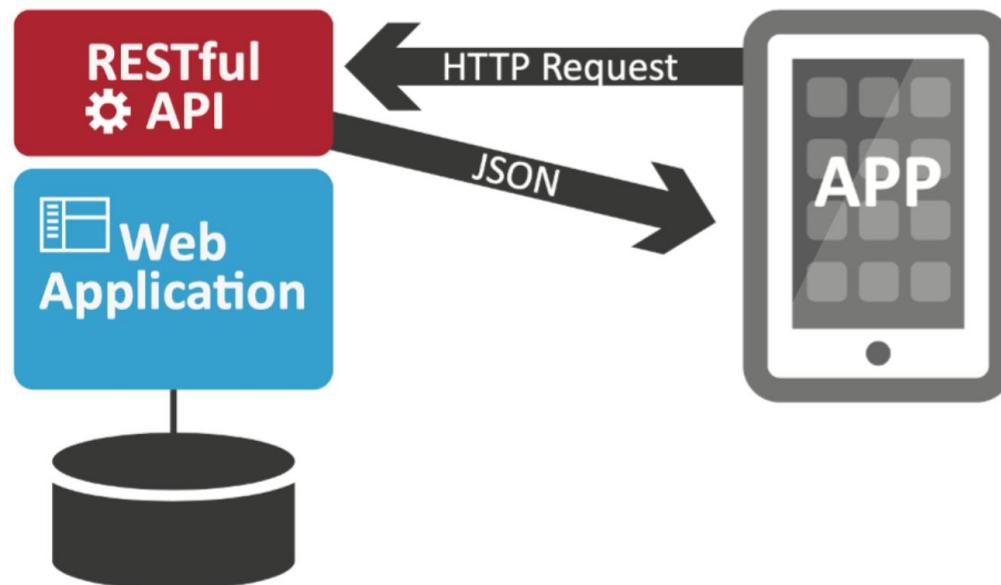
Introduzione ai web services



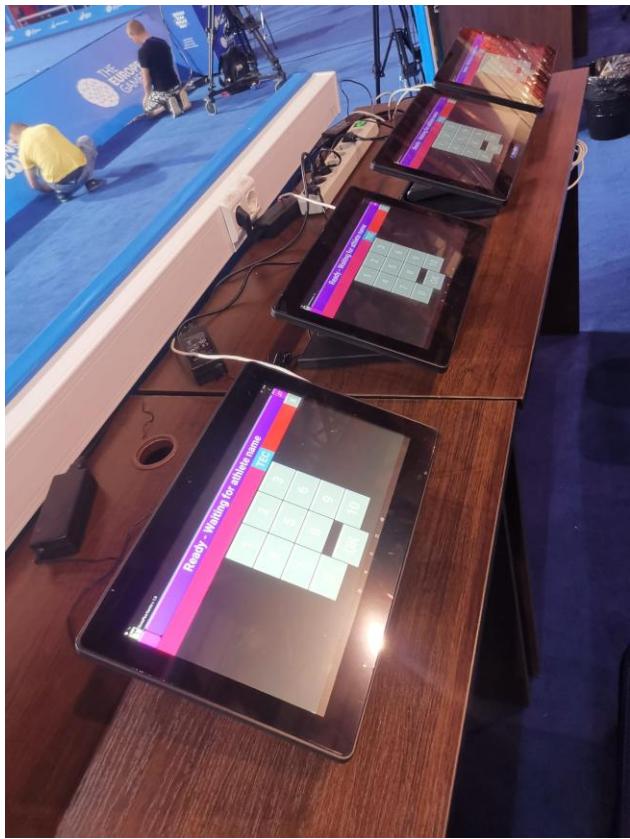
Introduzione ai web services

Anche se Ajax nasce in ambito web, allo stato attuale il client di un web service non deve necessariamente essere un browser, ma può essere anche :

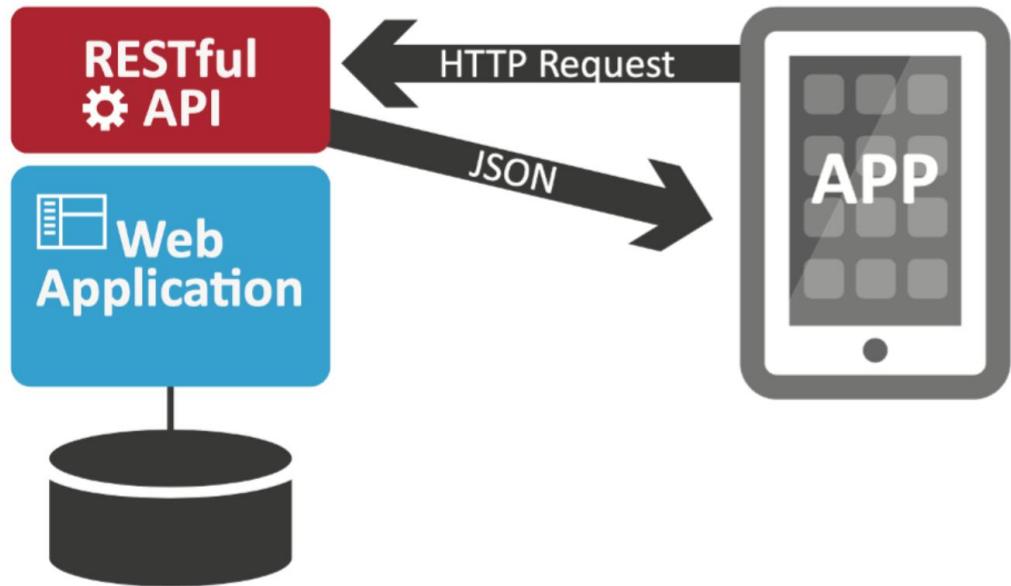
- **una normale applicazione desktop (ad esempio una applicazione C#)**
- **una app per smartphone (Android o IOS).**



Introduzione ai web services



Introduzione ai web services



Elementi base di AJAX

Asynchronous JavaScript And XML.

E' la tecnologia utilizzata dai browser (e non solo) per accedere ai dati esposti da un web server.

Lo scopo principale è quello di consentire l'aggiornamento dinamico dei dati contenuti in una pagina html senza dover ricaricare l'intera pagina con conseguente inutile spreco di risorse.

Con Ajax è possibile richiedere soltanto i dati necessari. Le applicazioni risultano così molto più veloci e la quantità di dati scambiati fra client e server si riduce notevolmente.

Elementi base di AJAX

Asynchronous JavaScript And XML.

E' la tecnologia utilizzata dai browser (e non solo) per accedere ai dati esposti da un web server.

Lo scopo principale è quello di consentire l'aggiornamento dinamico dei dati contenuti in una pagina html senza dover ricaricare l'intera pagina con conseguente inutile spreco di risorse.

Con Ajax è possibile richiedere soltanto i dati necessari. Le applicazioni risultano così molto più veloci e la quantità di dati scambiati fra client e server si riduce notevolmente.

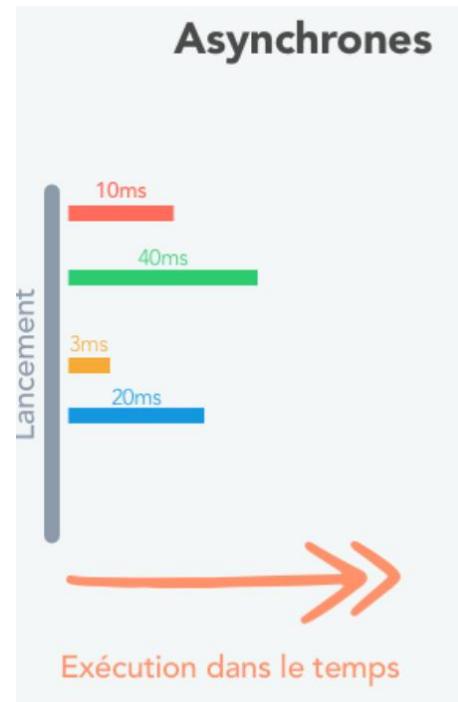


A dispetto del nome, la tecnologia AJAX oggi:

- oltre a javascript, è disponibile in quasi tutti gli ambienti di programmazione
- può scambiare dati in qualsiasi formato (XML, JSON, CSV o anche semplice testo)

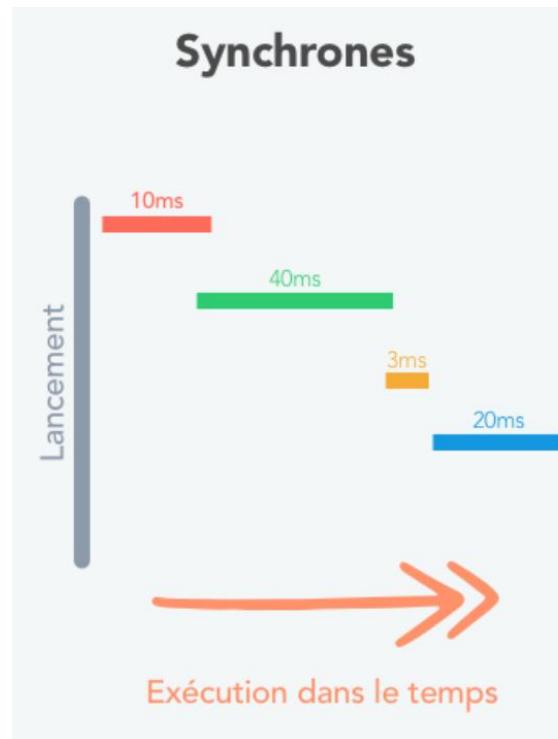
Elementi base di AJAX

Una caratteristica fondamentale di AJAX è che si tratta di una tecnologia **asincrona** nel senso che la richiesta viene inviata in background all'interno di un thread separato senza interferire con il comportamento della pagina. L'utente, una volta inviata la richiesta, può continuare ad interagire con l'interfaccia grafica anche mentre l'applicazione è in attesa dei dati.



Elementi base di AJAX

Concetto contrapposto al modello tradizionale di **comunicazione sincrona** tipica delle vecchie applicazioni web (le cosiddette applicazioni Web Form) in cui, in corrispondenza del submit, viene inviata una richiesta al server rimanendo poi in attesa del caricamento della nuova pagina.



Utilizzo dell'oggetto XMLHttpRequest

Un semplice esempio di utilizzo di Ajax potrebbe essere la scelta di un nuovo nickname in fase di creazione di un nuovo account su un sito web. In corrispondenza di ogni carattere digitato si invia una richiesta al server chiedendo se il nickname fino a quel momento inserito è valido oppure no.

A tale scopo può essere utilizzato l'evento `onChange` (o `OnKeyUp`) della Text Box.

- in corrispondenza di ogni carattere, java Script invia in tempo reale una richiesta al server
- Il server valuta se il nome fin'ora inserito è valido o no elaborando una risposta molto 'leggera'.
- In base alla risposta Java Script aggiorna opportunamente l'aspetto del textbox

Es. su gmail quando vogliamo verificare se un nome utente per la nuova mail è valido oppure no!

```
var richiesta = new XMLHttpRequest();
var url="controlla.php?parametro=" + encodeURIComponent(txtUsername.value);
// apro la connessione TCP con il server
richiesta.open("GET", url, true);

// le requestHeader possono essere assegnate solo DOPO l'apertura della connessione
richiesta.setRequestHeader(
    "Content-type", "application/x-www-form-urlencoded; charset=utf-8");

// funzione di callback da eseguire in corrispondenza della risposta
richiesta.onreadystatechange = aggiorna;
richiesta.send(null);
```

Parametri del metodo open

```
richiesta.open("GET", url, true);
```

1. Il metodo con cui inviare i parametri al server (GET / POST)
2. La url della risorsa richiesta. La url può essere espressa in due modi:
 - path relativo a partire dalla cartella corrente. Es `url="controllo.php"`
 - path assoluto a partire dalla cartella htdocs. Es `url="/5B/ese12/controllo.php"`La funzione **encodeURIComponent** consente di codificare eventuali caratteri speciali.
3. La modalità di esecuzione della send (true=asincrona, false=sincrona). Se si intende gestire una funzione di callback per la lettura della risposta, l'invio dovrà necessariamente essere asincrono.

Parametri del metodo open

```
richiesta.open("GET", url, true);
```

1. Il metodo con cui inviare i parametri al server (GET / POST)
2. La url della risorsa richiesta. La url può essere espressa in due modi:
 - path relativo a partire dalla cartella corrente. Es `url="controllo.php"`
 - path assoluto a partire dalla cartella htdocs. Es `url="/5B/ese12/controllo.php"`La funzione **encodeURIComponent** consente di codificare eventuali caratteri speciali.
3. La modalità di esecuzione della send (`true=asincrona, false=sincrona`). Se si intende gestire una funzione di callback per la lettura della risposta, l'invio dovrà necessariamente essere asincrono.

Parametri del metodo open

```
richiesta.open("GET", url, true);
```

1. Il metodo con cui inviare i parametri al server (GET / POST)
2. La url della risorsa richiesta. La url può essere espressa in due modi:
 - path relativo a partire dalla cartella corrente. Es `url="controllo.php"`
 - path assoluto a partire dalla cartella htdocs. Es `url="/5B/ese12/controllo.php"`La funzione **encodeURIComponent** consente di codificare eventuali caratteri speciali.
3. La modalità di esecuzione della send (true=asincrona, false=sincrona). Se si intende gestire una funzione di callback per la lettura della risposta, l'invio dovrà necessariamente essere asincrono.

Attributi da associare alla richiesta

```
richiesta.setRequestHeader(  
    "Content-type", "application/x-www-form-urlencoded; charset=utf-8");
```

Attributi da associare alla richiesta

```
richiesta.setRequestHeader(  
    "Content-type", "application/x-www-form-urlencoded; charset=utf-8");
```

- L'attributo **setRequestHeader** definisce il formato dei parametri da inviare al server.

Attributi da associare alla richiesta

```
richiesta.setRequestHeader(  
    "Content-type", "application/x-www-form-urlencoded; charset=utf-8");
```

- L'attributo **setRequestHeader** definisce il formato dei parametri da inviare al server.

```
richiesta.onreadystatechange = aggiorna;
```

Attributi da associare alla richiesta

```
richiesta.setRequestHeader(  
    "Content-type", "application/x-www-form-urlencoded; charset=utf-8");
```

- L'attributo **setRequestHeader** definisce il formato dei parametri da inviare al server.

```
richiesta.onreadystatechange = aggiorna;
```

Attributi da associare alla richiesta

```
richiesta.setRequestHeader(  
    "Content-type", "application/x-www-form-urlencoded; charset=utf-8");
```

- L'attributo **setRequestHeader** definisce il formato dei parametri da inviare al server.

```
richiesta.onreadystatechange = aggiorna;
```

- L'attributo **onreadystatechange** consente di definire un riferimento alla funzione JavaScript di callback che dovrà essere eseguita in corrispondenza del ricevimento della risposta.
- La funzione di callback **NON è obbligatoria**. Il server (nel caso ad es di comandi DML) può anche **NON inviare una risposta**, nel qual caso la funzione di callback deve essere omessa.

Attributi da associare alla richiesta

```
richiesta.send(null);
```

Attributi da associare alla richiesta

```
richiesta.send(null);
```

- Il metodo **send** consente di inviare la richiesta al server. Il parametro del metodo send vale:
 - **null** nel caso di richieste di tipo GET (come quella attuale)
 - nel caso delle richieste POST contiene l'elenco dei parametri in formato nome=valore scritti all'interno di una unica stringa e separati da & (oppure scritti in formato json o formData).

Poiché il client potrebbe inviare una nuova richiesta prima che sia giunta la risposta alla richiesta precedente, è buona regola **istanziare** un apposito oggetto **XMLHttpRequest** per ogni comunicazione, oppure inviare la nuova richiesta soltanto in corrispondenza del ricevimento della risposta precedente.

Gestione della risposta

La risposta testuale elaborata dal server viene restituita all'interno della proprietà **.responseText** dell'oggetto richiesta.

```
function aggiorna () {  
    if (richiesta.readyState==4 && richiesta.status==200)  
        alert(richiesta.responseText) ;
```

Gestione della risposta

La risposta testuale elaborata dal server viene restituita all'interno della proprietà `.responseText` dell'oggetto richiesta.

```
function aggiorna() {  
    if (richiesta.readyState==4 && richiesta.status==200)  
        alert(richiesta.responseText);
```

La funzione aggiorna() può essere richiamata più volte nel corso della comunicazione.
In corrispondenza delle varie chiamate il parametro readyState può assumere valori differenti :

- 0: request not initialized
- 1: server connection established
- 2: request received
- 3: processing request
- 4:** request finished and response is ready

Se la riposta è pronta (`readyState=4`) e lo stato è corretto, allora si può leggere il contenuto della risposta.

Gestione della risposta

La risposta testuale elaborata dal server viene restituita all'interno della proprietà `.responseText` dell'oggetto richiesta.

```
function aggiorna() {  
    if (richiesta.readyState==4 && richiesta.status==200)  
        alert(richiesta.responseText);
```

La funzione aggiorna() può essere richiamata più volte nel corso della comunicazione.
In corrispondenza delle varie chiamate il parametro readyState può assumere valori differenti :

- 0: request not initialized
- 1: server connection established
- 2: request received
- 3: processing request
- 4:** request finished and response is ready

Se la riposta è pronta (readyState=4) e lo stato è corretto, allora si può leggere il contenuto della risposta.

Gestione della risposta

La risposta testuale elaborata dal server viene restituita all'interno della proprietà `.responseText` dell'oggetto richiesta.

```
function aggiorna() {  
    if (richiesta.readyState==4 && richiesta.status==200)  
        alert(richiesta.responseText);
```

La funzione aggiorna() può essere richiamata più volte nel corso della comunicazione.
In corrispondenza delle varie chiamate il parametro readyState può assumere valori differenti :

- 0: request not initialized
- 1: server connection established
- 2: request received
- 3: processing request
- 4:** request finished and response is ready

Se la riposta è pronta (readyState=4) e lo stato è corretto, allora si può leggere il contenuto della risposta.

Gestione della risposta

La risposta testuale elaborata dal server viene restituita all'interno della proprietà `.responseText` dell'oggetto richiesta.

```
function aggiorna() {  
    if (richiesta.readyState==4 && richiesta.status==200)  
        alert(richiesta.responseText);
```

La funzione aggiorna() può essere richiamata più volte nel corso della comunicazione.
In corrispondenza delle varie chiamate il parametro readyState può assumere valori differenti :

- 0: request not initialized
- 1: server connection established
- 2: request received
- 3: processing request
- 4:** request finished and response is ready

Se la riposta è pronta (readyState=4) e lo stato è corretto, allora si può leggere il contenuto della risposta.

Gestione della risposta

La risposta testuale elaborata dal server viene restituita all'interno della proprietà `.responseText` dell'oggetto richiesta.

```
function aggiorna() {  
    if (richiesta.readyState==4 && richiesta.status==200)  
        alert(richiesta.responseText);
```

La funzione aggiorna() può essere richiamata più volte nel corso della comunicazione.
In corrispondenza delle varie chiamate il parametro readyState può assumere valori differenti :

- 0: request not initialized
- 1: server connection established
- 2: request received
- 3: processing request
- 4: request finished and response is ready

Se la riposta è pronta (readyState=4) e lo stato è corretto, allora si può leggere il contenuto della risposta.

Aggiornamento della pagina

Supponendo di usare un servizio lato server per il controllo di uno username e supponendo che tale servizio risponda :

- “OK” in caso di username valido
- “NOK” in caso di username non valido

lato client si può scrivere la seguente procedura di visualizzazione:

Aggiornamento della pagina

Supponendo di usare un servizio lato server per il controllo di uno username e supponendo che tale servizio risponda :

- “OK” in caso di username valido
- “NOK” in caso di username non valido

lato client si può scrivere la seguente procedura di visualizzazione:

```
function aggiorna () {
    if (richiesta.readyState==4 && richiesta.status==200) {
        var msg = $("#msg");
        var btn = $("#btnInvia");
        var risposta = richiesta.responseText;
        if (risposta.toUpperCase() == "OK") {
            msg.text("Nome valido");
            msg.css("color", "green");
            btn.prop("disabled", false);
        }
        else if (risposta.toUpperCase() == "NOK"){
            msg.text("Nome già esistente");
            msg.css("color", "red");
            btn.prop("disabled", true);
        }
        else
            alert("Risposta non valida \n"+risposta);
    }
}
```

Aggiornamento della pagina

Supponendo di usare un servizio lato server per il controllo di uno username e supponendo che tale servizio risponda :

- “OK” in caso di username valido
- “NOK” in caso di username non valido

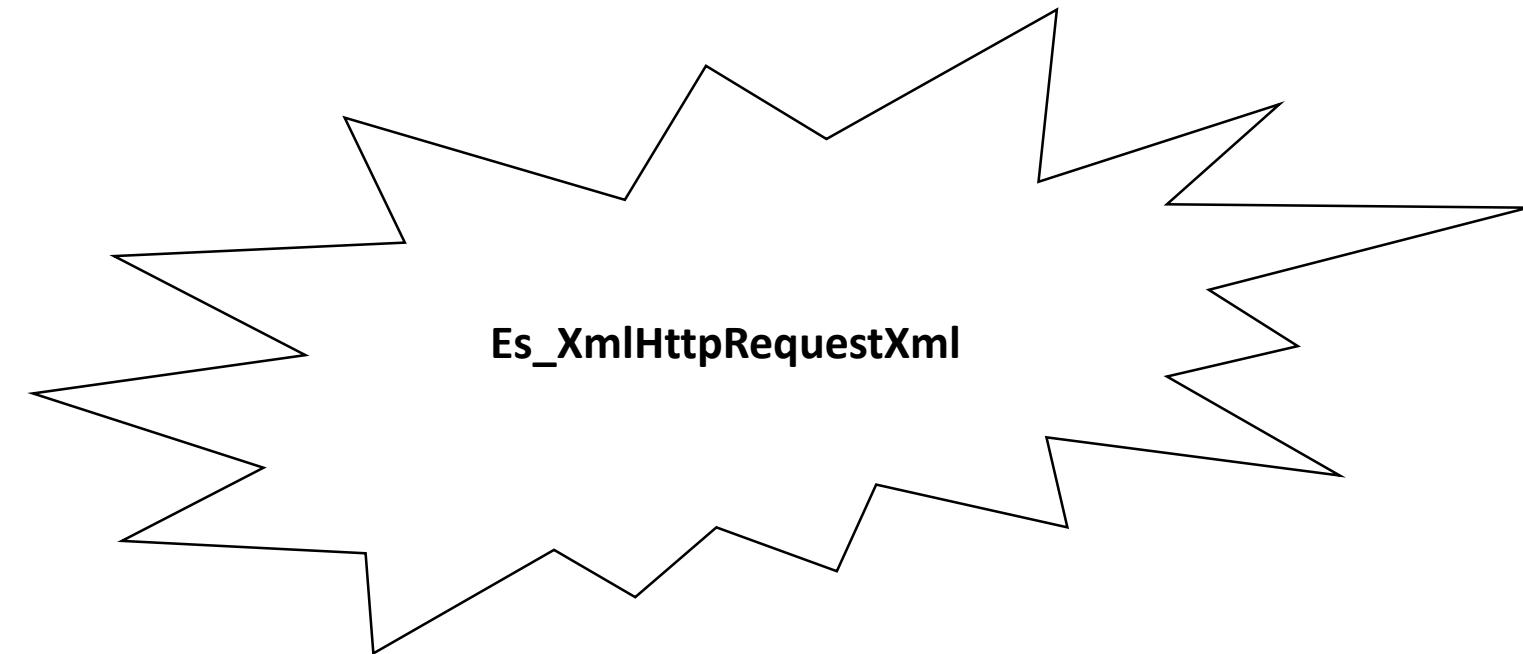
lato client si può scrivere la seguente procedura di visualizzazione:

```
function aggiorna () {
    if (richiesta.readyState==4 && richiesta.status==200) {
        var msg = $("#msg");
        var btn = $("#btnInvia");
        var risposta = richiesta.responseText;
        if (risposta.toUpperCase() == "OK") {
            msg.text("Nome valido");
            msg.css("color", "green");
            btn.prop("disabled", false);
        }
        else if (risposta.toUpperCase() == "NOK"){
            msg.text("Nome già esistente");
            msg.css("color", "red");
            btn.prop("disabled", true);
        }
        else
            alert("Risposta non valida \n"+risposta);
    }
}
```



Esercitazione con uso dell'oggetto XMLHttpRequest

Si richiede l'uso dell'oggetto XMLHttpRequest per l'ottenimento dei dati XML degli utenti provenienti dal webservices
<http://www.randomuser.com>



L'oggetto PROMISE

Si tratta di un oggetto mirato a facilitare la gestione dei processi asincroni.

La programmazione asincrona interviene nel momento in cui si eseguono operazioni “lente” che altrimenti bloccherebbero l’interfaccia grafica; ad esempio l’elaborazione grafica di una immagine oppure l’attesa di ricezione dati da un server esterno. In questi casi il sistema si prende **l’incarico** di invocare una funzione di callback al momento opportuno, e ritorna immediatamente il controllo all’applicazione.

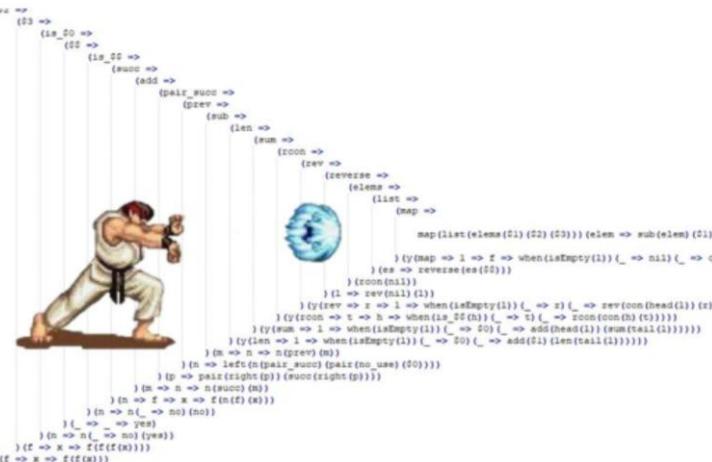
Sono frequenti anche i casi in cui un’operazione asincrona deve essere eseguita in coda ad un’altra operazione asincrona, dove ogni operazione dipende dal risultato dell’operazione precedente. In questi casi occorre annidare le callback una dentro l’altra con estensione del codice verso destra creando quella che è conosciuta come **"Piramide della sventura"** (**Pyramid of Doom**) orientata da sinistra verso destra.

L'oggetto PROMISE

Si tratta di un oggetto mirato a facilitare la gestione dei processi asincroni.

La programmazione asincrona interviene nel momento in cui si eseguono operazioni “lente” che altrimenti bloccherebbero l’interfaccia grafica; ad esempio l’elaborazione grafica di una immagine oppure l’attesa di ricezione dati da un server esterno. In questi casi il sistema si prende **l’incarico** di invocare una funzione di callback al momento opportuno, e ritorna immediatamente il controllo all’applicazione.

Sono frequenti anche i casi in cui un’operazione asincrona deve essere eseguita in coda ad un’altra operazione asincrona, dove ogni operazione dipende dal risultato dell’operazione precedente. In questi casi occorre annidare le callback una dentro l’altra con estensione del codice verso destra creando quella che è conosciuta come **"Piramide della sventura"** (**Pyramid of Doom**) orientata da sinistra verso destra.



L'oggetto PROMISE

```
function elaboraImmagine(img) {
  let promise = new Promise(function(resolve, reject) {
    ----- avvia elaborazione -----
    imgLibrary.onEnd(err, finalImage){ // elaborazione terminata
      if(err)
        reject(err)
      else
        resolve(finalImage);
    })
  })
  return promise;
}
```

Cos'è l'oggetto promise?

Definisce una PROMESSA che ha come unico parametro una function anonima alla quale vengono automaticamente iniettati due puntatori a funzione denominati **resolve** e **reject**

Terminata l'elaborazione, i codice interno della promise dovrà richiamare il metodo **resolve** in caso di terminazione corretta oppure il metodo **reject** in caso di errore

L'oggetto PROMISE

```
let request = elaboraImmagine(img)
request.then(function(finalImage) {
    console.log(finalImage)
})
request.catch(function(err) {
    console.log(err.message)
})
request.finally(function() {
    // close connection
})
```

Cioè la funzione **elaboraImmagine** istanzia una Promise e termina immediatamente restituendo al main la Promise istanziate. Il main dovrà salvare questa Promise all'interno di una variabile solitamente denominata **request**.

L'oggetto PROMISE

```
let request = elaboraImmagine(img)
request.then(function(finalImage) {
    console.log(finalImage)
})
request.catch(function(err) {
    console.log(err.message)
})
request.finally(function() {
    // close connection
})
```

Nel momento in cui la funzione `elaboraImmagine` richiama il metodo `resolve`, automaticamente la promise provvede a generare un evento `then()`, iniettandogli il risultato passato a `resolve`, cioè `finalImage`.

L'oggetto PROMISE

```
let request = elaboraImmagine(img)
request.then(function(finalImage) {
    console.log(finalImage)
})
request.catch(function(err) {
    console.log(err.message)
})
request.finally(function() {
    // close connection
})
```

Nel momento in cui la funzione **elaboraImmagine** richiama il metodo **reject**, automaticamente la promise provvede a generare un evento **catch()**, iniettandogli un oggetto err relativo all'errore verificatosi

L'oggetto PROMISE

```
let request = elaboraImmagine(img)
request.then(function(finalImage) {
    console.log(finalImage)
})
request.catch(function(err) {
    console.log(err.message)
})
request.finally(function() {
    // close connection
})
```

L'evento **finally()** viene generato sia in seguito del `then()`, sia in seguito del `.catch()`

Se il codice interno non richiama né il metodo **resolve** né il metodo **reject**, la Promise rimane "appesa", cioè non vengono generati gli eventi `.then()`, `catch()` e tantomeno `finally()`

L'oggetto PROMISE

```
let request = elaboraImmagine(img)
request.then(function(finalImage) {
    console.log(finalImage)
})
request.catch(function(err) {
    console.log(err.message)
})
request.finally(function() {
    // close connection
})
```

Ricapitolando...

usando la promise io posso far compiere un elaborazione dei dati (che può dura anche molto tempo) e specificare che in caso del richiamo di **resolve** venga generato l'evento **then** ed in caso di richiamo del metodo **reject** venga generato l'evento **catch**. Utilizzando le promise come si può notare io indico gli eventi ESTERNAMENTE alla mia promessa!!

Dopo **then/catch** verrà sempre generato l'evento **finally** (che per esempio potrei usare per la chiusura della connessione)

L'oggetto PROMISE

Se al posto di una variabile intermedia si utilizza il concatenamento, diventa possibile ad ogni promise associare più funzioni di callback in cascata in quanto ogni promise restituisce a sua volta un'altra promise

```
promise.then(function() { console.log(1); })
  .then(function() { console.log(2); })
```

Indicando delle then «a cascata» è possibile impostare una promise che ottiene un risultato dall'evento then e che sfrutta il risultato su una successiva callback, ecc.

Esempio di callback-hell

```
function getDataFromAPI1(callback) {
  setTimeout(() => {
    console.log("Dati da API 1 ottenuti");
    callback(null, "Dati API 1");
  }, 1000);
}

function getDataFromAPI2(data1, callback) {
  setTimeout(() => {
    console.log("Dati da API 2 ottenuti, utilizzando:", data1);
    callback(null, "Dati API 2");
  }, 1000);
}

function getDataFromAPI3(data2, callback) {
  setTimeout(() => {
    console.log("Dati da API 3 ottenuti, utilizzando:", data2);
    callback(null, "Dati API 3");
  }, 1000);
}
```

```
// Inizia il callback hell
getDataFromAPI1(function (err, data1) {
  if (err) {
    console.error("Errore API 1:", err);
    return;
  }

  getDataFromAPI2(data1, function (err, data2) {
    if (err) {
      console.error("Errore API 2:", err);
      return;
    }

    getDataFromAPI3(data2, function (err, data3) {
      if (err) {
        console.error("Errore API 3:", err);
        return;
      }

      console.log("Tutti i dati ottenuti:", data3);
    });
  });
});
```

Esempio di callback-hell

```
// Inizia il callback hell
getDataFromAPI1(function (err, data1) {
  if (err) {
    console.error("Errore API 1:", err);
    return;
  }

  getDataFromAPI2(data1, function (err, data2) {
    if (err) {
      console.error("Errore API 2:", err);
      return;
    }

    getDataFromAPI3(data2, function (err, data3) {
      if (err) {
        console.error("Errore API 3:", err);
        return;
      }

      console.log("Tutti i dati ottenuti:", data3);
    });
  });
});
```

Spiegazione:

1. La funzione `getDataFromAPI1` simula una chiamata API asincrona e invoca il **callback** al termine dell'operazione.
2. All'interno di ogni callback, viene invocata un'altra funzione asincrona, creando così un effetto di "piramide".
3. Questo porta a una struttura di codice annidata, nota come "**callback hell**" o "**Pyramid of Doom**", perché più funzioni si annidano, più il codice diventa complesso e difficile da gestire.

Esempio di callback-hell

```
// Inizia il callback hell
getDataFromAPI1(function (err, data1) {
  if (err) {
    console.error("Errore API 1:", err);
    return;
  }

  getDataFromAPI2(data1, function (err, data2) {
    if (err) {
      console.error("Errore API 2:", err);
      return;
    }

    getDataFromAPI3(data2, function (err, data3) {
      if (err) {
        console.error("Errore API 3:", err);
        return;
      }

      console.log("Tutti i dati ottenuti:", data3);
    });
  });
});
```

Spiegazione:

1. La funzione `getDataFromAPI1` simula una chiamata API asincrona e invoca il **callback** al termine dell'operazione.
2. All'interno di ogni callback, viene invocata un'altra funzione asincrona, creando così un effetto di "piramide".
3. Questo porta a una struttura di codice annidata, nota come "**callback hell**" o "**Pyramid of Doom**", perché più funzioni si annidano, più il codice diventa complesso e difficile da gestire.

Problemi:

Esempio di callback-hell

```
// Inizia il callback hell
getDataFromAPI1(function (err, data1) {
  if (err) {
    console.error("Errore API 1:", err);
    return;
  }

  getDataFromAPI2(data1, function (err, data2) {
    if (err) {
      console.error("Errore API 2:", err);
      return;
    }

    getDataFromAPI3(data2, function (err, data3) {
      if (err) {
        console.error("Errore API 3:", err);
        return;
      }

      console.log("Tutti i dati ottenuti:", data3);
    });
  });
});
```

Spiegazione:

1. La funzione `getDataFromAPI1` simula una chiamata API asincrona e invoca il **callback** al termine dell'operazione.
2. All'interno di ogni callback, viene invocata un'altra funzione asincrona, creando così un effetto di "piramide".
3. Questo porta a una struttura di codice annidata, nota come "**callback hell**" o "**Pyramid of Doom**", perché più funzioni si annidano, più il codice diventa complesso e difficile da gestire.

Problemi:

- Il codice diventa meno leggibile e difficile da mantenere.

Esempio di callback-hell

```
// Inizia il callback hell
getDataFromAPI1(function (err, data1) {
  if (err) {
    console.error("Errore API 1:", err);
    return;
  }

  getDataFromAPI2(data1, function (err, data2) {
    if (err) {
      console.error("Errore API 2:", err);
      return;
    }

    getDataFromAPI3(data2, function (err, data3) {
      if (err) {
        console.error("Errore API 3:", err);
        return;
      }

      console.log("Tutti i dati ottenuti:", data3);
    });
  });
});
```

Spiegazione:

1. La funzione `getDataFromAPI1` simula una chiamata API asincrona e invoca il **callback** al termine dell'operazione.
2. All'interno di ogni callback, viene invocata un'altra funzione asincrona, creando così un effetto di "piramide".
3. Questo porta a una struttura di codice annidata, nota come "**callback hell**" o "**Pyramid of Doom**", perché più funzioni si annidano, più il codice diventa complesso e difficile da gestire.

Problemi:

- Il codice diventa meno leggibile e difficile da mantenere.
- Ogni livello di annidamento introduce una maggiore complessità nel gestire eventuali errori.

Altro esempio di callback-hell

Questo codice funziona (con ogni callback annidata dentro all'altra), ma diventa molto difficile da leggere e mantenere man mano che aumenta il numero di operazioni asincrone.

```
function readDataFromDatabase(callback) {
    setTimeout(() => {
        console.log("Dati letti dal database");
        callback(null, "Dati");
    }, 1000);
}

function processData(data, callback) {
    setTimeout(() => {
        console.log("Dati processati");
        callback(null, "Dati processati");
    }, 1000);
}

function saveData(data, callback) {
    setTimeout(() => {
        console.log("Dati salvati");
        callback(null, "Successo");
    }, 1000);
}
```

```
// Callback hell
readDataFromDatabase((err, data) => {
    if (err) {
        console.error("Errore nel leggere i dati");
    } else {
        processData(data, (err, processedData) => {
            if (err) {
                console.error("Errore nel processare i dati");
            } else {
                saveData(processedData, (err, result) => {
                    if (err) {
                        console.error("Errore nel salvare i dati");
                    } else {
                        console.log("Operazione completata con successo: " + result)
                    }
                });
            }
        });
    }
});
```

Callback-Hell vs Promises

```
// Inizia il callback hell
getDataFromAPI1(function (err, data1) {
  if (err) {
    console.error("Errore API 1:", err);
    return;
  }

  getDataFromAPI2(data1, function (err, data2) {
    if (err) {
      console.error("Errore API 2:", err);
      return;
    }

    getDataFromAPI3(data2, function (err, data3) {
      if (err) {
        console.error("Errore API 3:", err);
        return;
      }

      console.log("Tutti i dati ottenuti:", data3);
    });
  });
});
```

```
// Uso delle Promises per evitare callback hell
getDataFromAPI1()
  .then(data1 => getDataFromAPI2(data1))
  .then(data2 => getDataFromAPI3(data2))
  .then(data3 => {
    console.log("Tutti i dati ottenuti:", data3);
  })
  .catch(err => {
    console.error("Errore:", err);
 });
```

Callback-Hell vs Promises

```
// Uso delle Promises per evitare callback hell
getDataFromAPI1()
  .then(data1 => getDataFromAPI2(data1))
  .then(data2 => getDataFromAPI3(data2))
  .then(data3 => {
    console.log("Tutti i dati ottenuti:", data3);
  })
  .catch(err => {
    console.error("Errore:", err);
});
});
```

Vantaggi:

- **Codice più leggibile e lineare:** Non c'è annidamento di funzioni, il che rende il codice più facile da capire e mantenere.
- **Gestione degli errori centralizzata:** Con `.catch()`, puoi gestire errori in un unico punto senza dover duplicare il codice di gestione degli errori per ciascuna operazione asincrona.

Callback-Hell vs Async-Await

Anche il costrutto **async/await** è un'alternativa molto potente e rende il codice ancora più leggibile, permettendo di scrivere codice asincrono come se fosse sincrono.

```
// Funzione asincrona che utilizza async/await
async function getAllData() {
  try {
    const data1 = await getDataFromAPI1();
    const data2 = await getDataFromAPI2(data1);
    const data3 = await getDataFromAPI3(data2);

    console.log("Tutti i dati ottenuti:", data3);
  } catch (err) {
    console.error("Errore:", err);
  }
}

// Chiamata della funzione principale
getAllData();
```

Spiegazione:

1. **async**: La parola chiave `async` definisce una funzione asincrona. In una funzione asincrona puoi usare `await`.
2. **await**: Aspetta che una **Promise** si risolva. Quando usi `await`, la funzione si ferma finché la **Promise** non restituisce un valore. Questo rende il codice simile a una sequenza sincrona.
3. **try/catch**: Gli errori possono essere gestiti facilmente con `try/catch`, in modo simile alla gestione degli errori sincrona.

Callback-Hell vs Async-Await

Anche il costrutto **async/await** è un'alternativa molto potente e rende il codice ancora più leggibile, permettendo di scrivere codice asincrono come se fosse sincrono.

```
// Funzione asincrona che utilizza async/await
async function getAllData() {
  try {
    const data1 = await getDataFromAPI1();
    const data2 = await getDataFromAPI2(data1);
    const data3 = await getDataFromAPI3(data2);

    console.log("Tutti i dati ottenuti:", data3);
  } catch (err) {
    console.error("Errore:", err);
  }
}

// Chiamata della funzione principale
getAllData();
```

Spiegazione:

1. **async**: La parola chiave `async` definisce una funzione asincrona. In una funzione asincrona puoi usare `await`.
2. **await**: Aspetta che una **Promise** si risolva. Quando usi `await`, la funzione si ferma finché la **Promise** non restituisce un valore. Questo rende il codice simile a una sequenza sincrona.
3. **try/catch**: Gli errori possono essere gestiti facilmente con `try/catch`, in modo simile alla gestione degli errori sincrona.

Vantaggi:

- **Codice più leggibile**: Il flusso del codice sembra sincrono, eliminando l'annidamento tipico delle callback o la complessità delle promesse concatenate.
- **Facilità nella gestione degli errori**: Puoi gestire gli errori in modo naturale con `try/catch`, rendendo il codice pulito e centralizzando la gestione degli errori.

Ritorniamo all'oggetto PROMISE...

```
let request = elaboraImmagine(img)
request.then(function(finalImage) {
    console.log(finalImage)
})
request.catch(function(err) {
    console.log(err.message)
})
request.finally(function() {
    // close connection
})
```

Ritorniamo all'oggetto PROMISE...

```
let request = elaboraImmagine(img)
request.then(function(finalImage) {
    console.log(finalImage)
})
request.catch(function(err) {
    console.log(err.message)
})
request.finally(function() {
    // close connection
})
```

Il codice precedente può anche essere scritto nel modo seguente, senza l'utilizzo della variabile intermedia **request**, concatenando il **.then()** e **.catch()** direttamente alla funzione principale.

```
elaboraImmagine(img)
.then(function(finalImage) {
    console.log(finalImage)
})
.catch(function(err) {
    console.log(err.message)
})
```

Concatenamento degli eventi

```
elaboraImmagine(img)
  .then(function(finalImage) {
    console.log(finalImage)
  })
  .catch(function(err) {
    console.log(err.message)
  })
```

Questa sintassi però non è del tutto equivalente alla precedente.

Sia `.then` che `.catch` restituiscono a loro volta una **nuova** promise che 'maschera' la promise principale alla quale possono essere associati un nuovo `.then()` ed un nuovo `.catch()` Per cui:

- 
- **Se si utilizza prima `.then` e dopo il `.catch`, il catch diventa gestore dell'errore sia rispetto alla promise principale, sia rispetto al codice interno del `then` e pertanto può visualizzare errori fuorvianti che 'mascherano' il vero motivo dell'errore.**
 - **Se si mette prima il `.catch` e dopo il `.then`, se il catch non introduce altri errori, il `then` viene eseguito in cascata**

Mentre il primo caso è ancora tollerabile, il secondo assolutamente NO.

Utilizzando una variabile intermedia request il problema non si pone.

Per cui gestire le promise mediante una variabile intermedia rimane comunque sempre preferibile.

Concatenamento degli eventi

```
elaboraImmagine(img)
  .then(function(finalImage) {
    console.log(finalImage)
  })
  .catch(function(err) {
    console.log(err.message)
  })
```

Questa sintassi però non è del tutto equivalente alla precedente.

Sia `.then` che `.catch` restituiscono a loro volta una **nuova** promise che 'maschera' la promise principale alla quale possono essere associati un nuovo `.then()` ed un nuovo `.catch()`. Per cui:

- **Se si utilizza prima `.then` e dopo il `.catch`, il catch diventa gestore dell'errore sia rispetto alla promise principale, sia rispetto al codice interno del then e pertanto può visualizzare errori fuorvianti che 'mascherano' il vero motivo dell'errore.**
- **Se si mette prima il `.catch` e dopo il `.then`, se il catch non introduce altri errori, il then viene eseguito in cascata**



Mentre il primo caso è ancora tollerabile, il secondo assolutamente NO.

Utilizzando una variabile intermedia request il problema non si pone.

Per cui gestire le promise mediante una variabile intermedia rimane comunque sempre preferibile.

Concatenamento degli eventi

```
elaboraImmagine(img)
  .then(function(finalImage) {
    console.log(finalImage)
  })
  .catch(function(err) {
    console.log(err.message)
  })
```

Questa sintassi però non è del tutto equivalente alla precedente.

Sia `.then` che `.catch` restituiscono a loro volta una **nuova** promise che ‘maschera’ la promise principale alla quale possono essere associati un nuovo `.then()` ed un nuovo `.catch()`. Per cui:

- **Se si utilizza prima `.then` e dopo il `.catch`, il catch diventa gestore dell'errore sia rispetto alla promise principale, sia rispetto al codice interno del `then` e pertanto può visualizzare errori fuorvianti che ‘mascherano’ il vero motivo dell'errore.**
- **Se si mette prima il `.catch` e dopo il `.then`, se il catch non introduce altri errori, il `then` viene eseguito in cascata**

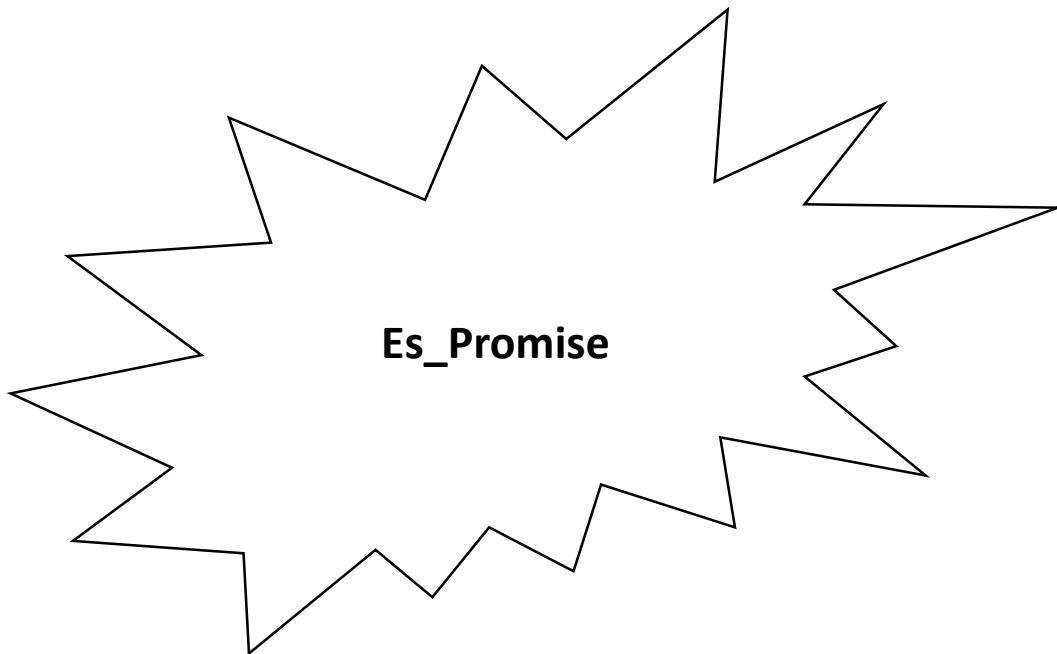
Mentre il primo caso è ancora tollerabile, il secondo assolutamente NO.

Utilizzando una variabile intermedia request il problema non si pone.

Per cui gestire le promise mediante una variabile intermedia rimane comunque sempre preferibile.

Esercitazione con uso dell'oggetto Promise

Si richiede l'uso dell'oggetto Promise per l'uso dei metodi **resolve/reject** ed il relativo richiamo degli eventi **then()** / **catch()**



Oggetti per l'invio di richieste AJAX

Per l'invio delle richieste ajax esistono diversi oggetti **ciascuno dei quali** restituisce al chiamante **SEMPRE** una **promise** che il chiamante potrà gestire all'interno del proprio codice.

`$.ajax()`

E' un metodo statico della libreria jQuery. (non presente nella libreria jquery **slim**) che rappresenta un wrapper dell'oggetto java script **XMLHttpRequest** per l'invio di una richiesta ajax ad un server.

Molto più semplice e flessibile rispetto ad XMLHttpRequest.

A differenza di **XMLHttpRequest**, `$.ajax()` è in grado di gestire **richieste multiple** verso una stessa risorsa con una stessa funzione di callback. In pratica passa un indice al server che lo rimanderà indietro in modo che la callback possa capire a quale elemento è riferita la risposta.

jQuery però gestisce le promise in modo leggermente diverso rispetto a javascript. Senza entrare nel dettaglio del funzionamento delle promise jQuery si può semplificare dicendo che le promise jQuery :

- Non generano gli eventi **then** e **catch**
- Al loro posto generano gli eventi **done** e **fail** simili ma non identici rispetto a quelli javascript

Oggetti per l'invio di richieste AJAX

Per l'invio delle richieste ajax esistono diversi oggetti **ciascuno dei quali** restituisce al chiamante **SEMPRE** una **promise** che il chiamante potrà gestire all'interno del proprio codice.

`$.ajax()`

E' un metodo statico della libreria jQuery. (non presente nella libreria jquery **slim**) che rappresenta un wrapper dell'oggetto java script **XMLHttpRequest** per l'invio di una richiesta ajax ad un server.

Molto più semplice e flessibile rispetto ad XMLHttpRequest.

A differenza di **XMLHttpRequest**, `$.ajax()` è in grado di gestire **richieste multiple** verso una stessa risorsa con una stessa funzione di callback. In pratica passa un indice al server che lo rimanderà indietro in modo che la callback possa capire a quale elemento è riferita la risposta.

jQuery però gestisce le promise in modo leggermente diverso rispetto a javascript. Senza entrare nel dettaglio del funzionamento delle promise jQuery si può semplificare dicendo che le promise jQuery :

- Non generano gli eventi **then** e **catch**
- Al loro posto generano gli eventi **done** e **fail** simili ma non identici rispetto a quelli javascript

La sintassi \$.ajax()

`$.ajax()` si aspetta come parametro un **json** di opzioni costituito dai seguenti campi:

```
let ajaxOptions={  
    url: "/url?nome=pippo",  
    type: "GET",          // default  
    data: { "nome": "pippo" },  
    contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default  
    dataType: "json",    // default  
    timeout : 5000,  
    async : true,         // default  
    success: function(data, textStatus, jqXHR) {  
        console.log(data)  
    },  
    error : function(jqXHR, textStatus, str_error){  
        if(jqXHR.status==0)  
            alert("connection refused or server timeout");  
        else if (jqXHR.status == 200)  
            alert("Errore Formattazione dati\n" + jqXHR.responseText);  
        else  
            alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);  
    },  
    username: "nome utente se richiesto dal server",  
    password: "password se richiesta dal server",  
}
```

La sintassi \$.ajax()

`$.ajax()` si aspetta come parametro un **json** di opzioni costituito dai seguenti campi:

```
let ajaxOptions={  
    url: "/url?nome=pippo", ← La risorsa  
    type: "GET",           // default  
    data: { "nome": "pippo" },  
    contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default  
    dataType: "json",     // default  
    timeout : 5000,  
    async : true,          // default  
    success: function(data, textStatus, jqXHR) {  
        console.log(data)  
    },  
    error : function(jqXHR, textStatus, str_error){  
        if(jqXHR.status==0)  
            alert("connection refused or server timeout");  
        else if (jqXHR.status == 200)  
            alert("Errore Formattazione dati\n" + jqXHR.responseText);  
        else  
            alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);  
    },  
    username: "nome utente se richiesto dal server",  
    password: "password se richiesta dal server",  
}
```

La sintassi \$.ajax()

`$.ajax()` si aspetta come parametro un **json** di opzioni costituito dai seguenti campi:

```
let ajaxOptions={  
    url: "/url?nome=pippo",  
    type: "GET",           // default ← Il metodo (GET/POST/ecc.)  
    data: { "nome": "pippo" },  
    contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default  
    dataType: "json",     // default  
    timeout : 5000,  
    async : true,          // default  
    success: function(data, textStatus, jqXHR) {  
        console.log(data)  
    },  
    error : function(jqXHR, textStatus, str_error){  
        if(jqXHR.status==0)  
            alert("connection refused or server timeout");  
        else if (jqXHR.status == 200)  
            alert("Errore Formattazione dati\n" + jqXHR.responseText);  
        else  
            alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);  
    },  
    username: "nome utente se richiesto dal server",  
    password: "password se richiesta dal server",  
}
```

La sintassi \$.ajax()

`$.ajax()` si aspetta come parametro un **json** di opzioni costituito dai seguenti campi:

```
let ajaxOptions={  
    url: "/url?nome=pippo",  
    type: "GET",          // default  
    data: { "nome": "pippo" },  
    contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default   
    dataType: "json",    // default  
    timeout : 5000,  
    async : true,        // default  
    success: function(data, textStatus, jqXHR) {  
        console.log(data)  
    },  
    error : function(jqXHR, textStatus, str_error){  
        if(jqXHR.status==0)  
            alert("connection refused or server timeout");  
        else if (jqXHR.status == 200)  
            alert("Errore Formattazione dati\n" + jqXHR.responseText);  
        else  
            alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);  
    },  
    username: "nome utente se richiesto dal server",  
    password: "password se richiesta dal server",  
}
```

Indica il formato con cui i parametri al server)

A proposito di content-type...

L'attributo **contentType**

Indica il formato con cui passare i parametri al server. Può assumere i seguenti valori:

```
contentType: "text/plain; charset=utf-8"  
contentType: "application/x-www-form-urlencoded; charset=utf-8"  
contentType: "application/json; charset=utf-8"      //stringa json  
contentType: "application/xml; charset=utf-8"      //stringa xml
```

Nel caso di chiamate GET eventuali parametri possono essere indifferentemente

- accodati alla url *oppure*
- inseriti come json all'interno del campo **data**

Se contentType : "application/x-www-form-urlencoded", i parametri passati a `$.ajax()` in formato JSON, **vengono automaticamente convertiti in formato urlencoded**. Esempio:

```
let json = {"user1": {"nome": "pippo", "hobbies": ["calcio", "pesca"] } }  
?user1[nome]=pippo&user1[hobbies][0]=calcio&user1[hobbies][1]=pesca
```

I parametri **POST** possono invece essere passati direttamente in un formato "application/json"

In questo caso, prima di inviare la richiesta, occorre **SERIALIZZARE** l'intero json.

A proposito di content-type...

In caso di GET (o metodo differente da GET) in definitiva posso gestire così...

```
function inviaRichiesta(method, url, parameters :{} ={}) { Show usages
    let contentType;
    if(method.toUpperCase() == "GET")
        contentType = "application/x-www-form-urlencoded; charset=utf-8";
    else{
        contentType = "application/json; charset=utf-8"
        parameters = JSON.stringify(parameters);
    }
    return $.ajax({
        "url": url,
        "data": parameters,
        "type": method,
        "contentType": contentType,
        "dataType": "json", // default
        "timeout": 5000, // default
    });
}
```

La sintassi \$.ajax()

`$.ajax()` si aspetta come parametro un **json** di opzioni costituito dai seguenti campi:

```
let ajaxOptions={  
    url: "/url?nome=pippo",  
    type: "GET",           // default  
    data: { "nome": "pippo" },  
    contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default  
    dataType: "json",     // default ← Indica come $.ajax() deve restituire i dati ricevuti dal server  
    timeout : 5000,  
    async : true,          // default  
    success: function(data, [textStatus], [jqXHR]) {  
        console.log(data)  
    },  
    error : function(jqXHR, textStatus, str_error){  
        if(jqXHR.status==0)  
            alert("connection refused or server timeout");  
        else if (jqXHR.status == 200)  
            alert("Errore Formattazione dati\n" + jqXHR.responseText);  
        else  
            alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);  
    },  
    username: "nome utente se richiesto dal server",  
    password: "password se richiesta dal server",  
}
```

L'attributo dataType

Indica il formato con cui `$_ajax()` deve restituire al chiamante i dati ricevuti dal server.

Può assumere i seguenti valori esprimibili **solamente** in modo diretto senza application/ davanti.

```
dataType : "text"  
dataType : "json"  
dataType : "xml"  
dataType : "html"  
dataType : "script"
```

- Scegliendo “text” la risposta viene restituita a `onSuccess()` così com’è, indipendentemente dal fatto che sia testo oppure json oppure xml.
- Scegliendo **json**, `$_ajax` provvede automaticamente ad eseguire il **parsing** dello stream json ricevuto, restituendo a `onSuccess()` un oggetto json. Idem per xml

La sintassi \$.ajax()

`$.ajax()` si aspetta come parametro un **json** di opzioni costituito dai seguenti campi:

```
let ajaxOptions={  
    url: "/url?nome=pippo",  
    type: "GET",           // default  
    data: { "nome": "pippo" },  
    contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default  
    dataType: "json",     // default  
    timeout : 5000,        ← 5000 ms identificano il timeout (tempo massimo per attendere la risposta del sever)  
    async : true,          // default  
    success: function(data, [textStatus], [jqXHR]) {  
        console.log(data)  
    },  
    error : function(jqXHR, textStatus, str_error){  
        if(jqXHR.status==0)  
            alert("connection refused or server timeout");  
        else if (jqXHR.status == 200)  
            alert("Errore Formattazione dati\n" + jqXHR.responseText);  
        else  
            alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);  
    },  
    username: "nome utente se richiesto dal server",  
    password: "password se richiesta dal server",  
}
```

La sintassi \$.ajax()

`$.ajax()` si aspetta come parametro un **json** di opzioni costituito dai seguenti campi:

```
let ajaxOptions={  
    url: "/url?nome=pippo",  
    type: "GET",           // default  
    data: { "nome": "pippo" },  
    contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default  
    dataType: "json",     // default  
    timeout : 5000,  
    async : true,          // default ← L'attributo async  
    success: function(data, [textStat] Impostando il valore false il metodo diventa sincrono, bloccando di fatto l'interfaccia grafica fino a  
        quando non sono arrivati i dati  
        console.log(data)  
    ),  
    error : function(jqXHR, textStatus, str_error){  
        if(jqXHR.status==0)  
            alert("connection refused or server timeout");  
        else if (jqXHR.status == 200)  
            alert("Errore Formattazione dati\n" + jqXHR.responseText);  
        else  
            alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);  
    },  
    username: "nome utente se richiesto dal server",  
    password: "password se richiesta dal server",  
}
```

La sintassi \$.ajax()

`$.ajax()` si aspetta come parametro un **json** di opzioni costituito dai seguenti campi:

```
let ajaxOptions={  
    url: "/url?nome=pippo",  
    type: "GET",           // default  
    data: { "nome": "pippo" },  
    contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default  
    dataType: "json",     // default  
    timeout : 5000,  
    async : true,          // default  
    success: function(data, textStatus, jqXHR) {  
        console.log(data) ← »success» viene richiamato in corrispondenza della risposta da parte del server  
    },  
    error : function(jqXHR, textStatus, str_error) { ← Callback richiamata in caso di errore  
        if(jqXHR.status==0)  
            alert("connection refused or server timeout");  
        else if (jqXHR.status == 200)  
            alert("Errore Formattazione dati\n" + jqXHR.responseText);  
        else  
            alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);  
    },  
    username: "nome utente se richiesto dal server",  
    password: "password se richiesta dal server",  
}
```

Uso di success / error

Il metodo **success**(data, textStatus , jqXHR)

Viene richiamato in corrispondenza della ricezione della risposta. In caso di content-type non testuale (ad esempio json o xml) **provvede automaticamente a parsificare la risposta ricevuta restituendo al chiamante l'object corrispondente**. Questa conversione automatica viene fatta in un “thread” separato e quindi alleggerisce la nostra applicazione..

textStatus indica lo stato in cui si è conclusa la XMLHttpRequest

jqXHR è un riferimento all'oggetto XMLHttpRequest sottostante utilizzato per inviare la richiesta

Il metodo **error**(jqXHR, textStatus, str_error)

In caso di errore, invece di richiamare la funzione **success**, viene richiamata la funzione **error**.

Questa funzione viene richiamata :

- **in caso di timeout** (es. se non ricevo risposta entro 5000 millisecondi)
- in corrispondenza della ricezione di un codice di **errore diverso da 200**
- in caso di ricezione di status==200 ma con un **oggetto json non valido** (se **dataType="json"**)
(ad esempio se il server va in syntax error e restituisce un messaggio di errore)

jqXHR è un riferimento all'oggetto XMLHttpRequest sottostante utilizzato per inviare la richiesta

textStatus indica lo stato in cui si è conclusa la XMLHttpRequest

Il terzo parametro rappresenta un msg di errore fisso legato al codice di errore

Creazione della funzione per l'invio della richiesta

```
const URL = "https://randomuser.me"

function inviaRichiesta(method, url, parameters={}) {

    let contentType;
    if(method.toUpperCase() == "GET")
        contentType = "application/x-www-form-urlencoded; charset=utf-8";
    else{
        contentType = "application/json; charset=utf-8"
        parameters = JSON.stringify(parameters);
    }

    let ajaxOptions={
        url: URL + url,
        type: method
        data: parameters,
        contentType: contentType
        dataType: "json", // default
        timeout : 5000,
    }

    return $.ajax(ajaxOptions) // ritorna una promise
}
```

Creazione della funzione per l'invio della richiesta

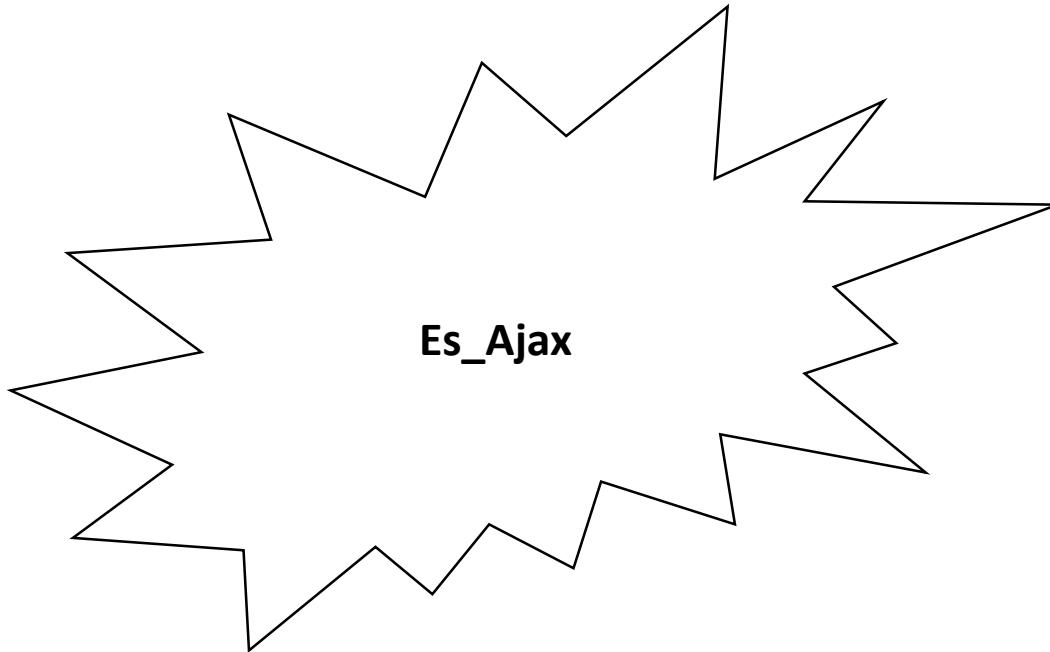
Chiamante

```
let request = inviaRichiesta("get", "/api?results=100")      // oppure
let request = inviaRichiesta("get", "/api", {"results":100})
request.fail(error);
request.done(function(data, test_status, jqXHR) {
    console.log(data);
});

function errore (jqXHR, test_status, str_error) {
    if(jqXHR.status==0)
        alert("connection refused or server timeout");
    else if (jqXHR.status == 200)
        alert("Errore Formattazione dati\n" + jqXHR.responseText);
    else
        alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);
}
```

Esercitazione con uso del metodo statico `$.ajax()`

Si richiede l'uso di `$.ajax()` per l'ottenimento di utenti contenuti di randomuser (richiedere il dato in XML)



Uso di Axios.js

Visto l'attuale declino di jQuery, **axios.js** (dal greco “**degno**”) rappresenta una degna alternativa a \$.ajax in direzione “vanilla” javascript. Restituisce una promise javascript con gli eventi .then e .catch

Al .**then** viene iniettata l'intera **http-response**. I dati ricevuti si trovano dentro **response.data**

I parametri GET vengono passati in **url-encoded** all'interno di un campo **params**

In alternativa è sempre possibile accodarli alla URL.

I parametri POST vengono passati in **json** all'interno di un campo **data**.

La serializzazione viene fatta automaticamente da axios.

```
function inviaRichiesta(method, url, parameters={}) {
  let config={
    "baseURL": "http://localhost:3000",
    "url": url,
    "method": method,
    "headers": {
      "Accept": "application/json",
    },
    "responseType": "json",
    "timeout": 5000
  }
  if(method.toUpperCase() === "GET"){
    config["params"] = parameters // plain object or URLSearchParams object
    config.headers["Content-Type"] = 'application/x-www-form-urlencoded; charset=utf-8'
  }
  else{
    config["data"] = parameters // Accept FormData, File, Blob
    config.headers["Content-Type"] = 'application/json; charset=utf-8'
  }
  return axios(config) // return a promise
}

function errore(err) {
  if(!err.response)
    alert("Connection Refused or Server timeout");
  else if (err.response.status == 200)
    alert("Formato dei dati non corretto : " + err.response.data);
  else
    alert("ServerError: " + err.response.status + " - " + err.response.data)
}
```

Uso di fetch

Metodo nativo javascript (2015) sostitutivo del vecchio XMLHttpRequest.

fetch() presenta un unico argomento obbligatorio, la URL della risorsa a cui accedere. E' possibile passare secondo parametro di **fetchOptions**, che sono più o meno le stesse di \$.ajax() e axios().

La chiamata va sempre a buon fine, cioè fetch restituisce sempre un oggetto **Response** indipendentemente dall'http status code della risposta (sia 200 oppure un qualunque codice di errore). Fetch genera errore soltanto in caso di errore di rete a seguito di timeout.

L' oggetto **Response** restituito da fetch espone:

- una proprietà **ok** impostata su **true** in caso di ok e su **false** per risposte diverse da 200–299

Espone inoltre

- una proprietà **status** contenente l'http response code
- una proprietà **statusText** contenente un http response message fisso

L'oggetto **Response** non contiene però direttamente l'effettivo corpo della risposta JSON (come nel caso di axios), ma per ottenere i dati veri e propri occorre richiamare un ulteriore metodo asincrono **response.json()** il quale restituisce una seconda **promise** che si risolve iniettando i dati veri e propri.

Uso di fetch

Metodo nativo javascript (2015) sostitutivo del vecchio XMLHttpRequest.

fetch() presenta un unico argomento obbligatorio, la URL della risorsa a cui accedere. E' possibile passare secondo parametro di **fetchOptions**, che sono più o meno le stesse di \$.ajax() e axios().

La chiamata va sempre a buon fine, cioè fetch restituisce sempre un oggetto **Response** indipendentemente dall'http status code della risposta (sia 200 oppure un qualunque codice di errore). Fetch genera errore soltanto in caso di errore di rete a seguito di timeout.

L' oggetto **Response** restituito da fetch espone:

- una proprietà **ok** impostata su **true** in caso di ok e su **false** per risposte diverse da 200–299

Espone inoltre

- una proprietà **status** contenente l'http response code
- una proprietà **statusText** contenente un http response message fisso

L'oggetto **Response** non contiene però direttamente l'effettivo corpo della risposta JSON (come nel caso di axios), ma per ottenere i dati veri e propri occorre richiamare un ulteriore metodo asincrono **response.json()** il quale restituisce una seconda **promise** che si risolve iniettando i dati veri e propri.

Esempio di richiesta con fetch

```
const _URL = "" // "http://localhost:1337"
async function inviaRichiesta(method, url="", params={}) {
  method = method.toUpperCase()
  if(method=="GET") url += "?" + new URLSearchParams(params)
  let options = {
    "method": method,
    "headers":{},
    "mode": "cors",      // default
    "cache": "no-cache", // default
    "credentials": "same-origin", // default
    "redirect": "follow", // default
    "referrerPolicy": "no-referrer", // default no-referrer-when-downgrade
    // riduce il timeout rispetto al default (6s) Non sembra possibile incrementarlo
    // "signal": AbortSignal.timeout(500)
  }
  if(method!="GET") {
    if(params instanceof FormData){
      options.headers["Content-Type"]="multipart/form-data;"
      options["body"]=params // Accept FormData, File, Blob
    }
    else{
      options.headers["Content-Type"]="application/json";
      options["body"] = JSON.stringify(params)
    }
  }
  try{
    const response = await fetch(_URL + url, options)
    if (!response.ok) {
      let err = await response.text()
      alert(response.status + " - " + err)
      //return false or undefined
    }
    else{
      let data = await response.json().catch(function(err){
        console.log(err)
        alert("response contains an invalid json")
        //return false or undefined
      })
      return data;
    }
  }
  catch{
    alert("Connection Refused or Server timeout")
    // return false or undefined
  }
}
let data = inviaRichiesta("POST", "https://example.com/answer", {param:42} )
if (data) { }
```

Esempio di richiesta con fetch

```
const _URL = "" // "http://localhost:1337"
async function inviaRichiesta(method, url="", params={}) {
    method = method.toUpperCase()
    if(method=="GET") url += "?" + new URLSearchParams(params)

    let options = {
        "method": method,
        "headers":{},
        "mode": "cors",      // default
        "cache": "no-cache", // default
        "credentials": "same-origin", // default
        "redirect": "follow", // default
        "referrerPolicy": "no-referrer", // default no-referrer-when-downgrade
        // riduce il timeout rispetto al default (6s) Non sembra possibile incrementarlo
        // "signal": AbortSignal.timeout(500)
    }
}
```

- 1) Notare il `.toUpperCase()` sul method. `fetch()` esige che i metodi siano scritti in **MAIUSCOLO** !
- 2) Eventuali parametri GET devono essere accodati alla URL nel seguente modo
`url = url + "?" + new URLSearchParams(params)`
- 3) Occhio alle **headers**, che vanno sempre scritte fra parentesi quadre con l'iniziale maiuscola
- 4) CORS requests may only use the HTTP or HTTPS **URL scheme**.
fetch non invia richieste con altri **URL schema** (per cui non è utilizzabile con il protocollo file://)
- 5) Notare che **statusText** non contiene il messaggio di errore impostato dal server, ma un testo fisso di errore http associato al codice di errore ricevuto. Per accedere al messaggio ricevuto dal server occorre utilizzare il metodo asincrono **response.text()** come nell'esempio precedente.
- 6) Da approfondire la gestione degli interceptors

Esempio di richiesta con fetch

```
const _URL = "" // "http://localhost:1337"
async function inviaRichiesta(method, url="", params={}) {
    method = method.toUpperCase()
    if(method=="GET") url += "?" + new URLSearchParams(params)

    let options = {
        "method": method,
        "headers":{},
        "mode": "cors",      // default
        "cache": "no-cache", // default
        "credentials": "same-origin", // default
        "redirect": "follow", // default
        "referrerPolicy": "no-referrer", // default no-referrer-when-downgrade
        // riduce il timeout rispetto al default (6s) Non sembra possibile incrementarlo
        //"signal": AbortSignal.timeout(500)
    }
    ...
}
```

Approfondimenti sulle fetchOptions

mode : **cors** = normale funzionamento CORS

no-cors = cioè senza resource sharing. Consente solo SAFE request

same-origin =. Consente solo chiamate sulla stessa origine

credentials: **omit** = non invia credenziali (cookies e HTTP Credential)

same-origin = invia le credenziali solo nelle same-origin request

include = invia le credenziali sia nelle same-origin request che nelle cross-origin

(il valore *include* è incompatibile con Access-Control-Allow-Origin = "*****")

Content-Type: Può assumere i valori `application/x-www-form-urlencoded`,

`application/json`, `text/plain`, `multipart/form-data`,

body : può contenere i seguenti oggetti: `String` (or `String Literal`), `URLSearchParams`, `FormData`, `File`, `Blob`, `ArrayBuffer`, `TypedArray`, `DataView`

Esempio di richiesta con fetch

```
const _URL = "" // "http://localhost:1337"
async function inviaRichiesta(method, url="", params={}) {
    method = method.toUpperCase()
    if(method=="GET") url += "?" + new URLSearchParams(params)

    let options = {
        "method": method,
        "headers":{},
        "mode": "cors",      // default
        "cache": "no-cache", // default
        "credentials": "same-origin", // default
        "redirect": "follow", // default
        "referrerPolicy": "no-referrer", // default no-referrer-when-downgrade
        // riduce il timeout rispetto al default (6s) Non sembra possibile incrementarlo
        // "signal": AbortSignal.timeout(500)
    }
}
```

referrerPolicy: L'intestazione http **Referer** è simile a **Origin** ma, mentre Origin contiene SOLO l'origine della richiesta (protocollo, dominio e porta), Referer contiene l'intero path della richiesta compresi i parametri. Al fine di preservare la privacy, le policy del Referer della richiesta possono definire i dati che possono essere inclusi / omessi. E' possibile anche omettere l'intera l'intestazione del Referer. Viceversa Origin non può essere esclusa

"**no-referrer**" – non invia mai il Referer.

"**unsafe-url**" – invia sempre l'url completo nel Referer, anche per richieste HTTPS→HTTP.

Purtroppo tra le **fetchOptions** no c'è un **timeout** che deve eventualmente essere gestito a mano.

Esempio di richiesta con fetch

```
const response = await fetch(_URL + url, options)
if (!response.ok) {
    let err = await response.text()
    alert(response.status + " - " + err)
    //return false or undefined
}
else{
    let data = await response.json().catch(function(err) {
        console.log(err)
        alert("response contains an invalid json")
        //return false or undefined
    })
    return data;
}
```

Metodi dell'oggetto response

Esiste un apposito metodo **asincrono** per ogni possibile oggetto ammesso all'interno del campo **body**:

- .text() : Returns the response as string data
- .json() : Returns the response as JSON
- .formData() : Return the response as FormData object (which stores key-value pairs of string data)
- .blob() : Returns the response as blob object (binary data along with its encoding)
- .arrayBuffer() : Return the response as ArrayBuffer (low-level representation of binary data)

Utilizzo di Async/Await

Rappresenta una modalità strutturata, alternativa al `.then` e `.catch` per gestire una promise restituita da una certa funzione. Consente di rendere ‘bloccante’ una funzione asincrona facendola sembrare sincrona.

Utilizzo di Async/Await

Rappresenta una modalità strutturata, alternativa al `.then` e `.catch` per gestire una promise restituita da una certa funzione. Consente di rendere ‘bloccante’ una funzione asincrona facendola sembrare sincrona.

Si riconsideri l’esempio iniziale di una funzione asincrona che restituisce una promise contenente il risultato di una elaborazione grafica di una immagine:

```
function elaboraImmagine(img) {
  let promise = new Promise(function(resolve, reject) {
    _lastLibrary.onEnd(err, finalImage) {
      if(err)
        reject(err)
      else
        resolve(finalImage);
    }
    return promise;
  })
}
```

Utilizzo di Async/Await

Rappresenta una modalità strutturata, alternativa al `.then` e `.catch` per gestire una promise restituita da una certa funzione. Consente di rendere ‘bloccante’ una funzione asincrona facendola sembrare sincrona.

Si riconsideri l’esempio iniziale di una funzione asincrona che restituisce una promise contenente il risultato di una elaborazione grafica di una immagine:

```
function elaboraImmagine(img) {
  let promise = new Promise(function(resolve, reject) {
    _lastLibrary.onEnd(err, finalImage) {
      if(err)
        reject(err)
      else
        resolve(finalImage);
    }
    return promise;
  })
}
```

Il chiamante, anziché utilizzare la solita gestione con `.then` e `.catch`,

```
let promise = elaboraImmagine(img)
promise.then(function (finalImage) { })
promise.catch( )
```

Utilizzo di Async/Await

Rappresenta una modalità strutturata, alternativa al `.then` e `.catch` per gestire una promise restituita da una certa funzione. Consente di rendere ‘bloccante’ una funzione asincrona facendola sembrare sincrona.

Si riconsideri l’esempio iniziale di una funzione asincrona che restituisce una promise contenente il risultato di una elaborazione grafica di una immagine:

```
function elaboraImmagine(img) {
  let promise = new Promise(function(resolve, reject) {
    _lastLibrary.onEnd(err, finalImage) {
      if(err)
        reject(err)
      else
        resolve(finalImage);
    }
    return promise;
  })
}
```

Il chiamante, anziché utilizzare la solita gestione con `.then` e `.catch`,

```
let promise = elaboraImmagine(img)
promise.then(function (finalImage) { })
promise.catch()
```

può utilizzare la seguente sintassi:

```
async function visualizzaImmagineElaborata() {
  let finaleImage = await elaboraImmagine(img)
```

Utilizzo di Async/Await

Gestione dell'errore

Se si intende gestire anche l'errore occorre utilizzare un TRY – CATCH che, alla fine, rende l'await abbastanza simile alla gestione .then .catch

```
async function visualizzaImmagineElaborata() {  
    try { let finaleImage = await elaboraImmagine(img) }  
    catch (err) { errore(err) }
```

In alternativa il .catch può essere gestito direttamente in coda alla funzione:

```
let finaleImage = await elaboraImmagine(img).catch(function(err) { ..... })
```

Utilizzo di Async/Await

Gestione dell'errore

Se si intende gestire anche l'errore occorre utilizzare un TRY – CATCH che, alla fine, rende l'await abbastanza simile alla gestione .then .catch

```
async function visualizzaImmagineElaborata() {  
    try { let finaleImage = await elaboraImmagine(img) }  
    catch (err) { errore(err) }
```

In alternativa il **.catch** può essere gestito direttamente in coda alla funzione:

```
let finaleImage = await elaboraImmagine(img).catch(function(err) { ..... })
```

La grandissima comodità dell'await sta nel fatto che i dati restituiti da await possono essere direttamente **ritornati** al chiamante (scrivendo il return nelle righe successive rispetto all'await stesso).

Utilizzo di Async/Await

Approfondimenti su await

await può essere utilizzata in due modi:

- 1) davanti al richiamo di una funzione asincrona che restituisce una promise (come nell'es precedente):

```
let finaleImage = await elaboraImmagine(file)
```

- 2) direttamente davanti ad una promise dichiarata all'interno della funzione stessa:

```
async function foo() {  
  let promise = new Promise(function(resolve, reject) {  
    setTimeout(() => resolve("done!"), 1000)  
  });  
  let result = await promise;  
  alert(result); // "done!"  
}
```

In entrambi i casi l'esecuzione si bloccherà sull'**await**, fino alla risoluzione della Promise.

Il metodo `Promise.all()`

Capita talvolta di dover eseguire parallelamente più operazioni asincrone.

Si supponga ad esempio di avere un DB di utenti, e di avere un vettore enumerativo di contatti contenente gli ID di alcuni di quegli utenti. Si vorrebbero richiedere singolarmente uno per uno tutti gli utenti aventi quegli ID e, al termine, eseguire una certa operazione conclusiva (inviare una risposta al client oppure eseguire una somma cumulativa su un campo numerico)

Il metodo `Promise.all()`

Capita talvolta di dover eseguire parallelamente più operazioni asincrone.

Si supponga ad esempio di avere un DB di utenti, e di avere un vettore enumerativo di contatti contenente gli ID di alcuni di quegli utenti. Si vorrebbero richiedere singolarmente uno per uno tutti gli utenti aventi quegli ID e, al termine, eseguire una certa operazione conclusiva (inviare una risposta al client oppure eseguire una somma cumulativa su un campo numerico)

Il client dovrebbe eseguire una richiesta per ogni ID. Gli approcci possibili sono due:

1. Eseguire le richieste in modo **annidato** ognuna all'interno del `then` della precedente. In corrispondenza della terminazione dell'ultima richiesta si esegue l'elaborazione.
Pesante sia dal punto di vista della scrittura sia dal punto di vista dell'esecuzione.
Migliorabile con l'utilizzo di `async- await`
2. Lanciare tutte le query in parallelo. Sorge però spontanea la domanda "come faccio ad aggregare le risposte?". L'ordine di esecuzione delle richieste non è detto che coincida con l'ordine di lancio.
Bisogna tenere un contatore esterno incrementato dopo ogni query. Quando il contatore raggiunge `vet.length` si risponde al client. Soluzione efficace ma abbastanza rudimentale

Il metodo `Promise.all()`

Capita talvolta di dover eseguire parallelamente più operazioni asincrone.

Si supponga ad esempio di avere un DB di utenti, e di avere un vettore enumerativo di contatti contenente gli ID di alcuni di quegli utenti. Si vorrebbero richiedere singolarmente uno per uno tutti gli utenti aventi quegli ID e, al termine, eseguire una certa operazione conclusiva (inviare una risposta al client oppure eseguire una somma cumulativa su un campo numerico)

Il client dovrebbe eseguire una richiesta per ogni ID. Gli approcci possibili sono due:

1. Eseguire le richieste in modo **annidato** ognuna all'interno del `then` della precedente. In corrispondenza della terminazione dell'ultima richiesta si esegue l'elaborazione.
Pesante sia dal punto di vista della scrittura sia dal punto di vista dell'esecuzione.
Migliorabile con l'utilizzo di `async- await`
2. Lanciare tutte le query in parallelo. Sorge però spontanea la domanda "come faccio ad aggregare le risposte?". L'ordine di esecuzione delle richieste non è detto che coincida con l'ordine di lancio.
Bisogna tenere un contatore esterno incrementato dopo ogni query. Quando il contatore raggiunge `vet.length` si risponde al client. Soluzione efficace ma abbastanza rudimentale

Un terzo approccio, simile al secondo ma molto più strutturato, è quello di lanciare tutte le richieste in parallelo. Poi, **invece di risolvere le singole Promise, le si accoda all'interno di un vettore**.

Si passa poi il vettore al metodo `Promise.all()` il quale **attende** che tutte le promise ricevute come parametro vengano risolte (con esito positivo o negativo).

- `Promise.all()` restituisce una nuova singola promise il cui `.then()` viene generato quando tutte le singole Promise sono state risolte con esito positivo.
- Se anche **una sola delle singole Promise** viene risolta con esito negativo, verrà generato il `.catch()` a cui viene iniettato l'errore generato dalla prima Promise che fallisce.

Il metodo `Promise.all()`

Capita talvolta di dover eseguire parallelamente più operazioni asincrone.

Si supponga ad esempio di avere un DB di utenti, e di avere un vettore enumerativo di contatti contenente gli ID di alcuni di quegli utenti. Si vorrebbero richiedere singolarmente uno per uno tutti gli utenti aventi quegli ID e, al termine, eseguire una certa operazione conclusiva (inviare una risposta al client oppure eseguire una somma cumulativa su un campo numerico)

Il client dovrebbe eseguire una richiesta per ogni ID. Gli approcci possibili sono due:

1. Eseguire le richieste in modo **annidato** ognuna all'interno del `then` della precedente. In corrispondenza della terminazione dell'ultima richiesta si esegue l'elaborazione.
Pesante sia dal punto di vista della scrittura sia dal punto di vista dell'esecuzione.
Migliorabile con l'utilizzo di `async- await`
2. Lanciare tutte le query in parallelo. Sorge però spontanea la domanda "come faccio ad aggregare le risposte?". L'ordine di esecuzione delle richieste non è detto che coincida con l'ordine di lancio.
Bisogna tenere un contatore esterno incrementato dopo ogni query. Quando il contatore raggiunge `vet.length` si risponde al client. Soluzione efficace ma abbastanza rudimentale

Un terzo approccio, simile al secondo ma molto più strutturato, è quello di lanciare tutte le richieste in parallelo. Poi, **invece di risolvere le singole Promise, le si accoda all'interno di un vettore**.

Si passa poi il vettore al metodo `Promise.all()` il quale **attende** che tutte le promise ricevute come parametro vengano risolte (con esito positivo o negativo).

- `Promise.all()` restituisce una nuova singola promise il cui `.then()` viene generato quando tutte le singole Promise sono state risolte con esito positivo
- Se anche **una sola delle singole Promise** viene risolta con esito negativo, verrà generato il `.catch()` a cui viene iniettato l'errore generato dalla prima Promise che fallisce.

Il metodo `Promise.all()`

Al `.then()` viene iniettato un **vettore enumerativo** contenente tutti gli oggetti iniettati alle singole Promise, **nello stesso ordine con cui state accodate le Promises**, indipendentemente dall'ordine di terminazione delle stesse.

```
let ids = [7, 12, 24, 52]
let promises = []

for (let id of ids)
    let promise = inviaRichiesta(.....)
    promises.push(promise)
}

Promise.all(promises).then((users) => {
    elaboraUsers(users);
});
```

Il metodo `Promise.all()`

Al `.then()` viene iniettato un **vettore enumerativo** contenente tutti gli oggetti iniettati alle singole Promise, **nello stesso ordine con cui state accodate le Promises**, indipendentemente dall'ordine di terminazione delle stesse.

```
let ids = [7, 12, 24, 52]
let promises = []

for (let id of ids)
    let promise = inviaRichiesta(.....)
    promises.push(promise)
}

Promise.all(promises).then((users) => {
    elaboraUsers(users);
});
```

- Se `inviaRichiesta`, per ogni richiesta, ritorna un singolo json, allora `users` sarà un semplice vettore enumerativo di json.
- Se `inviaRichiesta`, per ogni richiesta, dovesse ritornare un vettore enumerativo di record, allora `users` sarà un vettore enumerativo di vettori enumerativi, dove il vettore enumerativo di primo livello contiene al suo interno, cella per cella, i vari vettori enumerativi restituiti da ciascuna richiesta.

JSON-Server

JSON-Server è una piccola utility disponibile in ambiente nodejs in grado di rispondere a chiamate Ajax e restituire al chiamante il contenuto di un **file.json** memorizzato all'interno della cartella di lavoro.

Passi necessari all'installazione ed utilizzo di json-server

1. Installare NodeJS (<https://nodejs.org>)
2. Installare globalmente Json-Server:
`npm install -g json-server`
3. Scrivere nella cartella di progetto un file.json contenente il database dell'esercizio
4. Aprire un terminale nella cartella di lavoro
5. Se si è opportunamente configurata la variabile d'ambiente **PATH** con il percorso utente di nodejs, json-server potrà essere eseguito da qualsiasi cartella
6. Lanciare json-server in modo che utilizzi ad esempio il file db.json
`json-server --watch db.json`

L'opzione --watch indica che il file deve essere monitorato con continuità. Dalla versione 2024 questa opzione viene settata in automatico e dunque non è più necessaria

7. Il server risponde con un messaggio che conferma l'attivazione in localhost sulla porta **3000** e visualizza l'elenco delle risorse contenute all'interno del file db.json

Dopo aver abilitato la visualizzazione dei file nascosti aggiungere la **variabile di sistema per l'utente** (solo se usate il PC della scuola e quindi non siete amministratori) modificando la variabile PATH:

C:\Users**VOSTRONEUTENTE**\AppData\Roaming\npm\node_modules

N.B. da ricordare a MEMORIA poiché saranno richiesti durante le esercitazioni / compiti in classe

JSON-Server

All'interno del file db.json si può inserire dei gruppi di dati (detti **risorse**) che possono essere soltanto **vettori associativi** (json). Dalla versione 2024 i vettori enumerativi non sono più ammessi.

Per default json-server viene avviato soltanto sull'interfaccia `localhost` e dunque non sarà visibile sulla rete. Per aviarlo sull'interfaccia di rete occorre utilizzare l'opzione **--host**

```
json-server --host 10.0.1.2 --port 3000 --watch db.json
```

Esempio di file db.json contenente una unica risorsa denominata people

```
{ people: [  
  { "id": "abc1",  
    "nome": "Alfio",  
    "genere": "m",  
    "classe": "4B"  
  },  
  { "id": "abc2",  
    "nome": "Beatrice",  
    "genere": "f",  
    "classe": "4B"  
  },  
  { "id": "abc3",  
    "nome": "Carlo",  
    "genere": "m",  
    "classe": "1C"  
  }]  
}
```

Gestione degli ID

Dalla versione 2024, **id** deve tassativamente essere dichiarato come **stringa** di qualunque lunghezza.

Non sono più ammessi id numerici.

Se non è necessario ai fini del progetto, **ID può anche essere omesso**.

Per i record creati **dinamicamente** da codice (*e solo in questo caso*), ID viene creato automaticamente in modo univoco come stringa su 4 caratteri

JSON-Server

All'interno del file db.json si può inserire dei gruppi di dati (detti **risorse**) che possono essere soltanto **vettori associativi** (json). Dalla versione 2024 i vettori enumerativi non sono più ammessi.

Per default json-server viene avviato soltanto sull'interfaccia `localhost` e dunque non sarà visibile sulla rete. Per aviarlo sull'interfaccia di rete occorre utilizzare l'opzione `--host`

```
json-server --host 10.0.1.2 --port 3000 --watch db.json
```

Esempio di file db.json contenente una unica risorsa denominata people

```
{ people: [  
  { "id": "abc1",  
    "nome": "Alfio",  
    "genere": "m",  
    "classe": "4B"  
  },  
  { "id": "abc2",  
    "nome": "Beatrice",  
    "genere": "f",  
    "classe": "4B"  
  },  
  { "id": "abc3",  
    "nome": "Carlo",  
    "genere": "m",  
    "classe": "1C"  
  }]  
}
```

Gestione degli ID

Dalla versione 2024, **id** deve tassativamente essere dichiarato come **stringa** di qualunque lunghezza.

Non sono più ammessi id numerici.

Se non è necessario ai fini del progetto,
ID può anche essere omesso.

Per i record creati **dinamicamente** da codice (*e solo in questo caso*), ID viene creato automaticamente in modo univoco come stringa su 4 caratteri

JSON-Server

Richieste GET

Sono quelle più usate e consentono di richiedere dei dati al server

GET	/people	senza parametri restituisce un <u>vettore enumerativo</u> con tutte le persone
GET	/people/2	restituisce un <u>singolo record</u> relativo alla persona avente id=2
GET	/people?id=2	restituisce un <u>vettore enumerativo</u> lungo 1 contenente la persona indicata
GET	/people?genere=m&classe=4B.	Esegue una AND e restituisce un vettore di record

Richiesta di un singolo record sulla base dell'ID

L'ID può essere direttamente accodato alla risorsa dopo lo slash nel qual caso il server restituisce un *singolo record* relativo alla persona avente `id=2` oppure può essere passato in formato url-encoded dopo il punto interrogativo, nel qual caso il server restituisce un vettore enumerativo lungo 1

Richiesta di più record sulla base di altri campi

I parametri diversi dall'ID non possono essere accodati alla risorsa ma vanno concatenati alla url dopo il ? e suddivisi tra loro mediante una &, utilizzando cioè quello che si chiama formato **url-encoded**.

Ad esempio trovami tutti gli studenti di genere maschile della classe 4B

GET /people?genere=m&classe=4B. // Tutti i maschi della classe 4B

Per come è stata scritta inviaRichiesta(), i parametri, anziché essere concatenati in formato **url-encoded**, possono essere passati a inviaRichiesta() come terzo parametro in formato JSON (molto più comodo). inviaRichiesta() provvederà lei a **convertire** i parametri in url-encoded concatenandoli alla url.

```
inviaRichiesta("GET", "/people", {"genere": "m", "classe": "4B"}))
```

JSON-Server

Altre modalità di richiesta

In tutte le chiamate diverse dalla GET, il'ID può essere passato SOLO in coda alla risorsa:

POST /people aggiunge in coda a people il record passato come terzo **parametro**
{"nome":"pippo", "genere":"m", "classe":"2A"}

il campo ID può anche non essere assegnato, nel qual caso json-server provvede automaticamente ad assegnare al nuovo record un ID casuale **univoco** di tipo stringa. Restituisce dentro **response.data** l'intero record inserito, **comprendendo l'ID** assegnato automaticamente.

Non sembra possibile postare più record in una stessa chiamata.

DELETE /people/2 elimina il record avente id=2. Non ha parametri.

PUT /people/2 sostituisce il record id=2 con quello passato come **parametro**

PATCH /people/2 simile al precedente, ma aggiorna solo i singoli campi indicati

PATCH, a differenza di **PUT** (che sostituisce completamente l'intero record), aggiorna soltanto i campi passati come terzo parametro di tipo json, per cui non è necessario passare tutti i campi.

E' anche possibile passare uno o più campi non esistenti all'interno del record, campi che verranno automaticamente creati.

Nota

- Se in coda alla url permane un **?** oppure un **&** NON è un problema

JSON-Server

Campi annidati

Le ricerche possono anche accedere ad eventuali oggetti interni al json del database.

In caso di chiamate GET, la sintassi da utilizzare per passare a json-server un parametro annidato in formato url-encoded, **deve necessariamente essere la sintassi con il puntino :**



```
/people?location.country=italy
```

che in formato json diventa: `{"location.country": "italy"}`

Quindi la chiamata ajax dovrà espressamente avere il seguente formato:

```
inviaRichiesta( "GET", "/risorsa", {"location.country": "italy"} )
```

Viceversa, una chiamata del tipo

```
inviaRichiesta( "GET", "/risorsa", {location:{country:"italy"} } )
```

verrebbe convertita in url-encoded nel seguente modo, cioè con le parentesi quadre.

```
location[country]=italy
```

cioè sia \$.ajax che axios traducono i json annidati utilizzando le parentesi quadre e non il puntino.

json-server però non riconosce le quadre come strumento di indicizzazione ma richiede espressamente il puntino.

Quindi, in quest'ultimo caso, occorrerebbe eseguire una conversione manuale delle quadre con il puntino, utilizzando ad esempio il metodo statico `$.param`

```
let result = $.param(json).replaceAll("%5B", ".").replaceAll("%5D", "")  
%5B e %5D sono le parentesi quadre. In questo modo si ottiene il classico formato url-encoded con il puntino
```

```
nome=pippo&location.city=London&location.street=Carnaby+Street
```